SC^2: A Statistical Compression Cache Scheme

Angelos Arelakis Per Stenstrom
Chalmers University of Technology, Gothenburg, Sweden
{angelos,per.stenstrom}@chalmers.se

Abstract

Low utilization of on-chip cache capacity limits performance and wastes energy because of the long latency, limited bandwidth, and energy consumption associated with off-chip memory accesses. Value replication is an important source of low capacity utilization. While prior cache compression techniques manage to code frequent values densely, they trade off a high compression ratio for low decompression latency, thus missing opportunities to utilize capacity more effectively.

This paper presents, for the first time, a detailed design-space exploration of caches that utilize statistical compression. We show that more aggressive approaches like Huffman coding, which have been neglected in the past due to the high processing overhead for (de)compression, are suitable techniques for caches and memory. Based on our key observation that value locality varies little over time and across applications, we first demonstrate that the overhead of statistics acquisition for code generation is low because new encodings are needed rarely, making it possible to off-load it to software routines. We then show that the high compression ratio obtained by Huffman-coding makes it possible to utilize the performance benefits of 4X larger last-level caches with about 50% lower power consumption than such larger caches.

1. Introduction

On-chip cache memories are instrumental in tackling several performance and energy issues faced by contemporary and future processor architectures. First, they are key to bridge the growing speed-gap between memory and processors. Beyond this, they keep the demands in off-chip bandwidth within limits, as the bandwidth capacity is not growing in the same pace as the number of cores. Moreover, a larger on-chip cache can potentially bring down the total energy consumption, by reducing off-chip memory accesses. Hence, techniques that improve on-chip cache utilization are important.

Value replication – i.e., when the same value appears in multiple memory locations – is an important cause of poor memory utilization. Our earlier work shows that if each unique memory value is stored exactly once, compression ratios beyond 16X are possible [4], as only a subset of data values is used during program execution. Cache compression builds on this opportunity by encoding replicated values densely.

Assuming that data are uncompressed in memory, when a block is fetched into cache it must be first compressed and then decompressed when accessed or evicted from cache. Since decompression is on the critical memory-access path, previously proposed compression schemes trade off higher compression ratios for faster decompression. Frequent-pattern compression (FPC) [2] replaces frequent (pre-defined) data patterns, such as strings of zeros, with fixed-width codes. While such encoded words can be decompressed with low latency (a few cycles) the compression ratio is low, typically only 1.5X. C-pack [7] utilizes both static patterns and dictionaries, but the compression ratio is only marginally improved. On the other hand, Base-Delta Immediate (BDI) compression [19] compresses cache blocks by exploiting clustered value locality [26] but the compression ratio achieved is only slightly improved (~1.6X). However, BDI decompresses a block in a single cycle.

Our prior work on value-aware caches [4] shows that when cached data are compressed using statistical methods, such as Huffman coding, substantially higher compression ratios (4X) can be achieved. Unfortunately, that work did not explore how to transform that opportunity into practical cache designs. Therefore, whether statistics acquisition and the more complex (de)compression can be accommodated without affecting performance and energy consumption adversely remains an important and open question. In fact, the lack of such insights is the main reason to neglect such schemes as candidates for cache/memory compression despite their superior ratios.

This paper proposes *SC*^{2}, a practical cache design that leverages Huffman-based statistical compression to achieve significantly better cache utilization than previous proposals. Our detailed design-space exploration of statistical compression cache schemes reveals that it is possible to benefit from their better compression ratios without sacrificing performance even when cache capacity is sufficient. Our study makes the following contributions. First, we show that code-word generation needs to be done so rarely that its impact on overall performance is low despite its cost. Second, the latency of last-level caches is only modestly increased (about ten cycles) thanks to the pipelined decompression engine. More importantly, we show that is possible to get up to 4X more capacity, hence reducing off-chip accesses when cache capacity is insufficient. Therefore, SC^2 makes it possible to enjoy the performance benefits of 4X larger last-level caches at a power consumption that is about 50% lower than for 4X larger caches.

2. Motivation

The efficiency of the statistical compression cache schemes stems from the fact that *variable-length* codes are assigned to different data values based on their *probability of occurrence*. In the context of caches, some data values are more frequent

978-1-4799-4394-4/14/$31.00 © 2014 IEEE

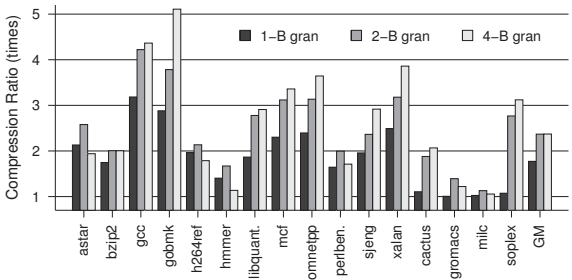

Figure 1: Cache (1MB) compression vs. value granularity.

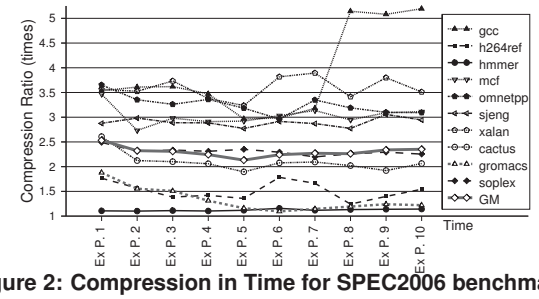

Figure 2: Compression in Time for SPEC2006 benchmarks.

than others. For example, 30% of the cache blocks in SPEC2K and server applications contain the value zero [9], while if each unique value is stored exactly once, a compression ratio beyond 16X is possible [4].

The potential downside of statistical compression is twofold. First, it requires a sampling phase in which statistics on the relative frequency of values are collected. This may involve too much processing to make such an approach feasible. Second, the processing involved in (de)compression is more complex and may end up on the critical memory-access path resulting in a too adverse impact on performance to make it useful.

Our proposed SC^2 scheme uses Huffman-coding [12] as it straightforwardly produces variable-length codes by constructing a Huffman tree starting from the least frequent values. An important dimension is the selection of data-type granularity (1, 2, 4, or 8 bytes), as it affects both the compressibility and decompression latency. Larin and Conte [16] consider 1-byte values resulting in a compression ratio (CR) of about 2X. On the other hand, in our previous work on value-aware caches [4], we establish an upper-bound of CR to 4X using 4-byte values and SPEC2K benchmarks. Figure 1 shows the CR for different granularities across a subset of the SPEC2006 applications. The encoding is constructed using the 1K most frequent values, while for further details regarding the experimental methodology the reader is referred to Section 4. Higher CR is obtained using 4- and 2-byte values for most applications, as the width of the generated encoding is smaller than the value representation width, contrary to 1-byte values. Small value replication is expected in 8-byte data values, thus they are not explored.

Statistical compression in caches requires two phases – sampling and compression – and codewords are constructed online between them. The question is how often sampling has to be done. Figure 2 shows that the compression ratio is not dramatically affected over time (measured in committed instructions and divided in execution points, *Ex. P*) for several representative SPEC2006 benchmarks. The Huffman coding is generated only once at the beginning of the simulation after a sampling phase of 80M committed instructions to warm up the last-level cache. The simulation duration corresponds to 10^9 committed instructions that would take on the order of a second to execute. These results confirm that sampling and code reconstruction are rarely needed. Our motivational data corroborate those of our value-aware cache study [4].

3. A Huffman-based Compressed Cache Design

We first provide an overview of the compressed cache design (Section 3.1) and then present in detail the structures needed and the operations carried out in Sections 3.2 through 3.7. SC^2 refers to our proposed Statistical Compression Cache scheme.

3.1. Overview

We consider a typical multi-core architecture with a multi-level cache hierarchy where compression is applied to the last-level cache (LLC). Our goal is to improve the utilization of the LLC to bring down the number of off-chip accesses, which can potentially improve performance and energy efficiency. Another goal is to minimize the performance impact of decompression for statistical compression schemes. We do this by an effective decompression process and by leveraging the latency tolerance capabilities of modern out-of-order execution cores.

As depicted in Figure 3, there are two main processes that manage the Huffman-based compressed cache: Sampling and Compression. First, the sampling phase takes place, followed by code-generation and eventually compression, where the cache content is compressed using the generated coding. As sampling and code generation can be carried out sufficiently rarely, code generation can be off-loaded to software.

During sampling, the frequency of all uniquely stored data values in the LLC is stored in the *Value-Frequency Table (VFT)*. When the sampling phase terminates, the VFT state is frozen and used by a software routine to generate the codewords. Then, the compression phase can start immediately (left part of Figure 3). Sampling and compression can also run at the same time. This may occur when the cache compressibility with the current encoding is not sufficient, as statistical compression methods are data dependent. Our design allows smooth transitions between coding versions, which is detailed further in Section 3.7.

Before a block is inserted into the LLC, the *Huffman Compressor (HuC)* attempts to compress it value-by-value (4-byte granularity). A value is kept uncompressed if a codeword does not exist for it, in which case a unique codeword (determined dynamically at code generation) is attached in front of it to distinguish it from compressed values during decompression. The (un)compressed values (words) within the block are concatenated in the original order and form the compressed block. As uncompressed values are expanded, a block is eventually compressed only if its size is smaller than the original.

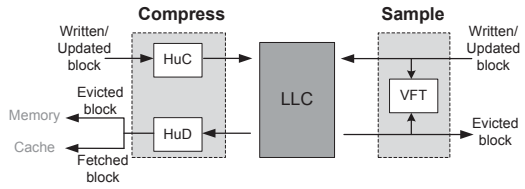

Figure 3: Main structures of the compressed cache

Table 1: Tag-array area cost for a 4-MB cache.

	Baseline		Compressed (2x)		Compressed (4x)	
	bits/tag	Total(KB)	bits/tag	Total(KB)	bits/tag	Total(KB)
addr	30	240	30	480	30	960
v, d, lru	6	8+8+32	7 (lru:+1)	16+16+80	8 (lru:+2)	32+32+192
c	0	0	1	16	1	32
index	0	0	10	160	10	320
Total	36	288	48	768	49	1568

Blocks that are fetched or evicted must be first decompressed in the *Huffman Decompressor (HuD)*, whose latency must be sufficiently short to only marginally affect performance. Next, we describe the structures that support the sampling and the (de)compression processes outlined above.

3.2. Cache Structures

Like previously proposed compressed cache designs, our approach is to store a larger number of compressed memory blocks in the original cache blockframe. Similar to prior arts [10, 2], tags are decoupled from the data store, allowing more tags to be associated with each blockframe and additional meta data. As shown in Figure 2, many applications exhibit a steady-state compression ratio close to three or more. This suggests, at least, to triple the number of tags associated with each set to take advantage of the compression potential. Of course, more tags introduce area overheads and may impact the cache hit time adversely.

Hosting multiple blocks in the same cache blockframe opens the question how to locate them. Unlike in FPC[2] and BDI [19], where the cache set is divided into fixed-sized segments (typically 8-bytes), in SC^2 a compressed block can be placed at any byte position in the data-array of the set. To do this, an index field must be associated with each tag to be able to locate the compressed block. For example, the index is 10 bits assuming a 16-way set associative cache with 64-byte blocks. This indirection introduces an extra delay that corresponds to the propagation delay of a decoder. A *c* bit is also needed per tag to indicate whether the block is compressed.

Table 1 summarizes the tag-store sizes for a 16-way 4-MB cache (64-byte blocks), with and without compression, assuming 48-bit physical addresses. The data store is the same for all cache designs with 64-K entries but the tag store contains 64-K, 128-K and 256-K entries in the baseline, the 2x and 4x (compressed) cache designs, respectively. For example, each tag is increased by 11 bits (due to the 'c' bit and index) and 2 bits due to extra LRU-bits when 4x more tags are used. The overall overhead of the tag store in the 4-MB baseline cache is 6.5% and in the compressed cache it goes up modestly to 15.7% (2x tags), 22.3% (3x tags) and 27.5% (4x tags).

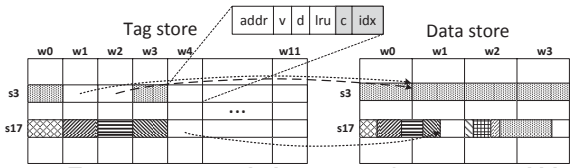

Figure 4: Tag structure and placement of compressed blocks.

3.3. Free-space Management and Replacement Policy

Another change of the conventional cache structure relates to the management of free space in the cache blockframe, which also requires modifications of the replacement policy. In a typical cache with LRU, a victim can be unambiguously selected. In SC^2, it may happen that several (possibly non-LRU blocks) must be replaced to free up contiguous space for a fetched block. The following scenarios define the free-space management actions that we propose and evaluate:

1. A new compressed block is *larger* than the replaced one: Adjacent compressed blocks must be evicted and their respective tags are invalidated to make room for the fetched block. This may cause eviction of MRU blocks but we find that it happens rarely, as no performance degradation is noticed. The same action is taken if the size of a modified compressed block becomes larger.
2. A new compressed block has the *same* size as the replaced one. In this case, no further action is needed.
3. A new compressed block is *smaller* than the replaced one, or a modified block becomes smaller than before: The freed-up space is allocated to the adjacent blocks if they are invalid. We also explore benefits related to compacting compressed blocks to release more contiguous space allocated to the LRU block. Contrary to ECM [5], compaction runs in the background after the new (or updated) block has been written.
4. In the beginning of the compression phase (transition from no compression to compression), the cache content is incrementally compressed. During this period, if an uncompressed block is about to be evicted, an attempt is first made to compress it instead of immediately evicting it.

Figure 4 depicts the tag store and data store in SC^2. In this figure, four uncompressed blocks can be saved in one set when compression is off (see s3 at figure) and 12 compressed blocks can be saved when it is on, thanks to the extra tags (3x tags). The tags are always used in a specific order and are associated with the compressed blocks by setting the index field. When compression is off, tags w0, w3, w6 and w9 are only used, while the extra tags in-between point to the end of each blockframe (e.g., w1 and w2 point to the end of the first blockframe). When compression is on, an attempt will be first made to compress the block pointed to by tag w0. If successful, the next tags' indexes, w1 and w2, will be updated based on the compressed block's size, otherwise w0 becomes the LRU. For example, in set 17 (s17), associated tags and compressed blocks share the same pattern. Tag w4 is invalid and points to the end of the compressed block w3.

3.4. The Sampling Phase

During the sampling phase, every cache access is monitored to keep track of the number of unique data values, i.e. the number of times a particular value is replicated in the cache. This information is maintained using a small set-associative cache called *Value-Frequency Table* (VFT) by saving pairs of a value and a counter. These counters estimate the frequency distribution of values in the cache and is required by the Huffman coding generation algorithm to provide the final encoding. The counter width (in bits) is determined by the maximum number of times a given value appears in the cache ($\lceil log_2(\frac{Cache\ size}{4\ bytes})\rceil$). Smaller counters can be used too but when one of them saturates, all the counts must be rescaled, i.e., divided by 2 (shift right by one) resulting in count-precision loss.

The VFT is accessed using the M least significant bits of the value as an index, where M is determined based on the size and associativity of the VFT. If a new value must be added in a full VFT set, the least frequently encountered value is evicted.

Three cache operations trigger an access to the VFT:

1. **Block fetch:** The VFT is looked up for every word (value) of the fetched block. On a VFT hit, the corresponding counter is incremented, while on a VFT miss the missed value is inserted in the VFT.
2. **Block update:** The counts of the updated block's values are incremented in the VFT, while the counts of the old values are decremented. As we assume inclusion, old values are available. New values are inserted in the VFT.
3. **Block eviction:** The counts of the evicted block's (either clean or dirty) values are decremented in the VFT.

3.5. Huffman Codeword Generation

When fast decoding is required and a large alphabet is used (on the order of 1-K VFT values in SC^2), canonical Huffman coding [22] is more efficient than the classic one because generated codewords of one length are consecutive binary numbers represented by as many bits as their length determines (*numerical sequence property*). Canonical codewords are generated starting with a codeword of the smallest length (say L) and assigning the value 0 represented by L bits. A codeword of length x is determined by the last codeword of length *x-1* using the formula C_x=$2*(C_{x-1}+1)$ (1) if there are valid codewords for length *x-1*; otherwise, formula C_x=$2*C_{x-1}$ (2) is used. Codeword generation requires the symbols to be sorted based on their corresponding Huffman code length, which is determined by the Huffman tree using the VFT.

Section 2 shows that sampling and code-generation is needed infrequently, thus SC^2 constructs the code in software. The Huffman codeword lengths are elegantly calculated using the min-heap data structure [20]. A heap stores a binary tree in an array without using pointers: If a tree node is stored at array position i, its children are stored at positions $2*i+1$ and $2*i+2$. The coding is generated in the following steps:

1. VFT counter values (or counts) are saved in one array ("Cnt") in VFT order. An extra array ("Heap"), which implements a heap, saves the indexes of the array "Cnt" in increasing order (min-heap) with respect to counter values ($T(N) = O(Nlog_2N)$, N: number of values). An additional pointer is initialized to the end of the heap.
2. The Huffman tree is constructed in many iterations by sifting down the heap. The final tree is stored in the "Heap" and "Cnt" arrays. In each iteration, the heap becomes smaller and the released heap position is devoted to the Huffman tree (for one of the intermediate nodes), which is gradually created ($T(N) = O(Nlog_2N)$). Each iteration:
   i) sifts down the heap. The discarded element is saved.
   ii) adds the counters of the previously discarded element and the one pointed to by the new "Heap" root element and saves the result in the last heap position;
   iii) updates the counters of these two elements (added in Huffman tree) with the current heap pointer so that they point to their parent node; and
   iv) updates the "Heap" root to point to the root of the tree, sorts the heap and updates the heap pointer so that the heap becomes smaller by one position.
3. For every element of the "Cnt", calculate the Huffman codeword length (cL) which equals the number of pointer jumps until reaching the Huffman tree root ($T(N) = O(Nlog_2N)$).
4. Sort the VFT values in the order of ascending codeword lengths ($T(N) = O(N)$).

The canonical Huffman codewords are eventually generated using formulas (1) and (2). In addition, the contents of the De-LUT, the Offset and the First Codewords (FCW) (see Figure 6b), which are needed by the decompressor and discussed thoroughly in Section 3.6, are generated. Figure 5 depicts the steps required to build the Huffman tree. At the end of this process, the "Heap" and the "Cnt" contain the Huffman tree.

Finally, the total number of values not present in the VFT, which is determined by subtracting the cumulative counts of the VFT values from the total number of cache values, is also used in the code generation as an alphabet symbol.

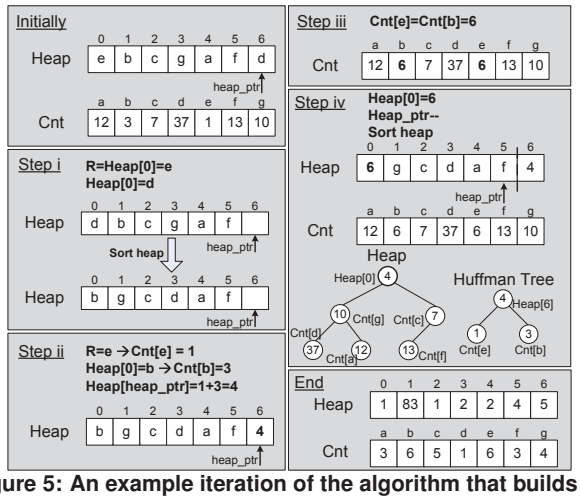

Figure 5: An example iteration of the algorithm that builds the Huffman tree using the Heap.

The generated codeword is the unique code attached in front of every uncompressed value, and its length depends on the value frequency distribution; the more unique values the cache contains, the smaller this codeword is. Alternatively, a static mask associated to every tag indicating whether a word is compressed could be used. However, this adds a fixed 2-byte overhead per tag, assuming 64-byte blocks.

3.6. The Compression Phase

During (de)compression, the following actions are taken in response to processor reads and writes (see Figure 3).

1. **Miss:** To keep compression off of the critical memory-access path, the uncompressed block is first supplied to the upper cache level. In parallel, the Huffman Compressor (HuC) compresses that same block before it is fetched into the LLC. The compressed block is placed in the cache blockframe at the position determined by the tag index. To make space for the fetched block, one of the four actions described in Section 3.3 is taken. The replaced dirty blocks are decompressed (if compressed) before being evicted.
2. **Write hit:** Before the block is updated, it is compressed and the new size is compared to the old one. The latter is calculated by subtracting two indexes: $T_{i+1}.idx - T_i.idx$, assuming $T_i.idx$ points to the old block. This happens off of the critical memory-access path. One of the four actions in Subsection 3.3 is taken to free up space.
3. **Read hit:** The block to be read is decompressed.

Read hits are critical as blocks must be first decompressed before responding to the processor. The bit-sequence of a compressed block i is decompressed by the HuD. The beginning and the end of it are pointed by tag indexes $T_i.idx$ and $T_{i+1}.idx - 1$. Compression is not on the critical memory-access path, but the decompressor must be fast to have the least impact on execution time.

Huffman Compressor: The Huffman Compressor (HuC) is depicted in Figure 6a. It is a small Lookup Table (LUT) that saves the canonical codeword (cw) and its length (cL), and it has the same number of entries as the VFT. The contents of both cw and cL are produced during code generation. cL is necessary because codewords have variable lengths but may be padded with zeros when saved in the LUT. There is also a limit on the codeword length. Through experimentation, we have found that lengths exceeding 19 bits are never used and in most of the cases 16 to 17 bits suffice. If a generated codeword is larger than the maximum length, it is not used and the respective value is stored uncompressed.

Huffman Decompressor: The Huffman Decompressor (HuD) is depicted in Figure 6b. We use the same Huffman Decompressor as proposed for the value-aware cache [4]. The elegant numerical sequence property of canonical Huffman codewords aids in quickly allocating a valid codeword and retrieving the corresponding value by accessing only one small LUT, i.e., the De-LUT. The VFT values are saved in the De-LUT in code-length ascending order during code generation.

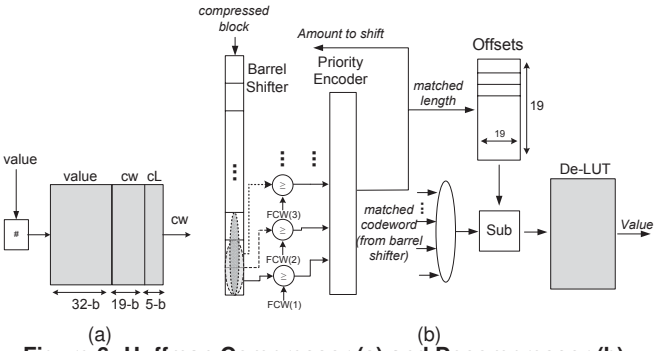

Figure 6: Huffman Compressor (a) and Decompressor (b).

```
BS: Barrel Shifter,    L: matched length,    mcw: matched codeword
UncCode: The code attached to every uncompressed value
START
while (values_found < 16) // 16 values per compressed block
   if (BS(18) ≥ FCW(1) AND BS(18 downto 17) < FCW(2)) then
      L = 1
      mcw = BS(18)
   else if (BS(18 downto 17) ≥  FCW(2) AND BS(18 downto 16) < FCW(3) ) then
      L = 2
      mcw = BS(18 downto 17)
   ..
   else if (BS(18 downto 0) ≥ FCW(19)) then
      L = 19
      mcw = BS (18 downto 0)

   if (mcw ≠ UncCode)
      BS << L
      idx = mcw - Offsets[L]
      Value = DeLUT[idx]
   else // next 32 bits are the Value
      empty BS and insert the bits after the next 32

   values_found++
END
```

Figure 7: Decompression process in pseudocode.

De-LUT and VFT have the same number of entries.

Although the values are stored in consecutive locations of the De-LUT, canonical codewords of different lengths are not consecutive numbers as they are generated based on the formulas of subsection 3.5. For example, if the last codeword of length 1 is $(0)_2$, then the first codeword of length 2 will be $(10)_2$. However, the corresponding value of the codeword $(10)_2$ is at location 1 of the De-LUT and not at location 2. An offset, which is calculated at code generation time, must be subtracted from the codeword. The codeword is padded with zeros before entering the subtracter. The N least significant bits of the difference are used as an address to the De-LUT, assuming 2^N De-LUT entries. The decompression process is explained using the pseudocode of Figure 7. First Codeword (FCW) is the first canonical Huffman codeword that was generated for a particular length during code generation.

3.7. Encoding Transition

While statistical compression is efficient, it is data-dependent. Despite the infrequent need for code regenerations as shown in Section 2, SC^2 uses a mechanism to support smooth transitions from one Huffman encoding to another. First, it uses a structure to monitor cache compressibility (continuously or periodically) by tracking the compressed block size of every new or updated block to compute the compression ratio (CR). At the end of a monitoring period, CR is compared against two

thresholds TH_l and TH_h and the following decisions are made: a) $< TH_l$ compression is de-activated, b) $> TH_h$ compression continues with current encoding, c) otherwise it continues with current encoding but at the same time, sampling is turned on.

After a new sampling, a new encoding is generated. To avoid flushing the cache that may contain data compressed with a previous encoding, an extra decompressor is used to allow two simultaneous encodings to carry out incremental recompression; whenever an old compressed block is decompressed, it is recompressed using the new encoding. As the compressor and decompressor are independent units, one HuC and two HuD are enough. This mechanism also requires an extra bit per tag for each simultaneously used encoding to determine the appropriate decompressor (0.7% extra tag overhead for a 4MB cache for two encoding-status bits). The energy consumption due to the extra decompressors is only slightly affected, as we later show in Section 5.3.

Given these additional hardware requirements, our design can smoothly tolerate encoding transitions. When two encodings are already in use and a third one is generated, the oldest coding is discarded by resetting the respective decompressor. Beforehand, all the dirty compressed blocks that use that encoding must be decompressed and evicted. These are located by a flash operation, similar to flash invalidation [11], using the dirty and encoding status bits. The selected blocks, which are likely dead, are buffered and eventually written back to memory and invalidated. As will be shown in Section 5.1, this is a rare event so this operation is not performance critical and is off of the critical memory access path. As an alternative, one can use a technique similar to DRAM-refresh; a block with an old encoding is decompressed and re-compressed (with the new encoding) once in a while so that by the time a third encoding is generated, it is guaranteed that no block is compressed with the old encoding.

4. Experimental Setup

We evaluate our proposed Huffman-based compressed cache using the cycle-accurate GEM5 simulator (version 2) [6] in syscall emulation using the classic memory system.

Simulated systems. We simulate both single-core and multi-core systems. The single-core system is important for two reasons. First, we use it to establish the efficiency of the Huffman-based compressed cache by using individual applications in isolation, shielding them from interference from others in a multi-core setting. Second, we use the characterization of each individual application derived from single-core simulations to systematically construct multiprogrammed workload mixes used in the multi-core simulations.

The baseline configurations are summarized in Table 2. We model a three-level cache hierarchy which resembles the ones used in recent Intel architectures, with private L1I/D and L2 caches per core, and an L3 cache (LLC), shared by all 8 cores in our multi-core configurations. All caches use 64-byte blocks. Cache latencies are estimated using CACTI [18]. The shared LLC is assumed to have 8 banks (one per core). In single-core simulations, we disable prefetching to explore the pure impact of compression. We enable it later to investigate how simple stride-based prefetching [13] interacts with our SC^2. In the multi-core simulations, prefetching is enabled. All single-core (BL1, BL2, BL3) and multi-core (MCBL1, MCBL2, MCBL3) baselines differ in LLC size and latency.

Table 2: Baseline configuration.

Single-core configurations: BL1/BL2/BL3	
Cores	1 core, X86-64, out-of-order, issue width 4, 3GHz
LQ:SQ, ROB, RAS	64:36, 168, 16
L1I/L1D	32 KB, 8-way, 2 cycles, 4 MSHR
L2	256 KB, 8-way, 5 cycles, 32 MSHR
L3 (BL1/BL2/BL3)	1/2/4 MB, 10/12/17 cycles, 8-way, 32 MSHR
Multi-core (8 cores) configurations: MCBL1/MCBL2/MCBL3	
Shared L3	4/8/16 MB, 20/28/37 cycles, 8 banks, 16-way, 32 MSHR
Memory & Bus	Mem: 200 cycles (66ns) - Bus: 32-byte width, 1GHz

Table 3: Huffman-based SC^2 configuration.

Single-core configurations: Huff-2x/-3x/-4x	
L3 (Huff-2x)	1MB (2x tags) / 8-w / 10 cycles
L3 (Huff-3x)	1MB (3x tags) / 8-w / 10 cycles
L3 (Huff-4x)	1MB (4x tags) / 8-w / 10 (+1) cycles
Multi-core configurations: MC-Huff-2x/-3x/-4x	
L3 (MC-Huff-2x)	4MB (2x tags) / 16-w / 20 cycles
L3 (MC-Huff-3x)	4MB (3x tags) / 16-w / 20 (+1) cycles
L3 (MC-Huff-4x)	4MB (4x tags) / 16-w / 20 (+1) cycles
Compressor / Decompressor	
Decompression lat.	8 / 14 cycles
Compression lat.	6 cycles
Compression enabled	20M committed instr. after simpoint start
Coding generation	Once per simpoint (at 20M instr.)
VFT / Compressor / DeLUT	7KB / 7KB / 4KB

In the SC^2 configurations, statistical compression is applied to the LLC, whose physical size is the same as the LLC of BL1 (1MB) in single-core, and of MCBL1 (4MB) in multi-core. The goal is to increase the LLC capacity, without increasing the physical size, and explore whether and to what extent we can improve performance, or in fact keep the same performance when the footprint of the application (or application mixes) fits in the uncompressed cache. The other two baselines (BL2-BL3, MCBL2-MCBL3) which have twice and four times more LLC capacity than SC^2, respectively, are used as reference points to estimate whether our design approaches their performance by achieving better utilization of the LLC.

Table 3 summarizes the configurations that employ our Huffman-based SC^2. The simulated system only differs from the baselines in the LLC (L3), which in this case faithfully models our SC^2 design. We consider three design points with 2x, 3x or 4x more tags. Based on our CACTI models, when 4x more tags are used, an extra cycle is added to the hit time of the LLC in the single-core configuration, whereas in the multi-core system this also applies to the configuration with 3x tags. The indirection to locate a compressed block corresponds to the propagation delay of a 9-to-512 decoder in the single-core configurations and a 10-to-1024 decoder in the multi-core ones, which was measured to 0.14ns and 0.16ns respectively using the Synopsys Design Compiler (F-2011.09-SP3).

Based on a detailed design analysis using CACTI and Synopsys, assuming a 32-nm technology node, the decompressor

can be pipelined in three stages (conservative stage delay of 0.25ns). The decompression latency is 8 clock cycles if the critical word is in the first half of the block and 14 cycles if it is in the second half, assuming a bus width of 32 bytes. The compressor is modeled in CACTI as a 7-KB, 8-way set-associative cache using two read/write ports with an estimated delay of 0.18ns. Using a conservative delay of 0.22ns, a block can be compressed in 6 clock cycles. Compression is performed using a 4-byte value granularity based on our findings in Section 2. The VFT is 7 KB (4-KB values and 3-KB counters) and is configured exactly the same as the compressor. Both compressor and VFT actions do not appear on the critical memory-access path, but small buffers are placed in front of them to avoid congestion. This can especially happen in the VFT in case a burst of values needs to be updated.

In the multi-core system, each LLC bank is associated with a compressor and a decompressor to maintain the throughput. However, a single VFT is enough for sampling in spite of sharing the LLC among several applications. In this case, we simply double the VFT size to fit more values, which also doubles the compressor's and DeLUT sizes but does not affect the (de)compression latency. This greatly simplifies management of the (de)compression processes, as caches do not necessarily have to be flushed upon a context switch.

Compressing the shared cache of a multi-core system does not affect coherence. Assuming that each tag contains coherence state and sharer vectors, coherence requests that are served 1) with LLC data are not affected by decompression latency if it partly/totally overlaps with invalidations/inv-acknowlegdement transactions for write misses, whilst 2) compression has no impact on requests without LLC data (cache-to-cache transfers). Finally, concurrent reads that hit in LLC are not affected either as there are multiple decompressors.

We implement the code generation algorithm in C++. The encoding is generated in approximately 1 millisecond when running on our simulated BL1 system (1K values in the VFT), and 1.3 milliseconds on the MCBL1 (2K values in the VFT). We have taken this latency into account when establishing the total execution time. The transition period needed in action 4 of Section 3.3 is set to 10M committed instructions.

Workloads. We use all integer and 8 floating point (FP) applications of the SPEC2006 benchmark suite with the reference input. The rest of the FP applications could not run on GEM5 due to unimplemented syscalls. For the multi-input integer applications *perlbench, bzip2, gcc, gobmk, hmmer, h264ref* and *astar* we use the inputs *splitmail, liberty, g23, score2, nph3, baseline* and *biglakes*, respectively. We compile the benchmarks with gcc using the optimization flag O2. We use simpoints [23] to determine representative simulation phases of the applications (10-15 simpoints per application, 250M committed instructions each).

The multi-core system is evaluated using multiprogrammed workloads comprising application mixes constructed from the aforementioned benchmarks, using the methodology described in Section 5.2. Simulation starts after fast-forwarding of 5 billion committed instructions and warming up the caches for another 500M committed instructions. The detailed simulations run until every application has committed 250M instructions. When a benchmark completes 250M instructions, we collect the statistics and keep it running to stress shared resources until all of them complete 250M instructions. A similar simulation methodology is used in related works [21, 27].

Metrics. We use speedup of execution, MPKI (Misses per Kilo Instructions), BPKI (Bytes per Kilo Instructions) and compression ratio (CR). For every application, these statistics are calculated using the weighted average (every simpoint has a weight). The number of committed instructions used to calculate MPKI is 250M. BL1 is the reference for speedup in the single-core simulations. In the multi-core simulations, we take MCBL1 as a reference and we calculate the weighted speedup $\sum \frac{IPC_i}{IPC_i^{alone}}$ [24]. IPC_i is the IPC of core i in shared resources and IPC_i^{alone} is the IPC of core i running alone.

The compression ratio is calculated for every application using 2 simpoints but of large duration (i.e., 10^9 instructions for each simpoint) to explore whether compression deteriorates in longer execution phases than using only 250M instructions. For BL1 and for each application (both simpoints), snapshots of the cache data store are taken every 100M committed instructions. Huffman encoding is generated only for the first snapshot of each simpoint, assuming 4-byte values. Using this encoding, we estimate the compression ratio for each snapshot by calculating the average compression in every set ($Compression = \frac{Original\ set\ size}{Compressed\ set\ size}$) and then across cache sets. Figure 2 in Section 2 is derived based on this methodology. However, in Figure 1 the coding is constructed per snapshot.

Prior work. We simulate FPC [2] and BDI [19] to compare with state-of-the-art compression approaches. The FPC and BDI configurations have a decompression latency of 5 and 1 cycle, respectively, while the segment size is selected to 1 byte to investigate their full compression potential.

5. Results

5.1. Single-core Evaluation

We first explore what impact an increased cache capacity has on cache performance (measured in MPKI) and on the total execution time for each application. Figures 8 and 9 summarize the results for the baselines BL1, BL2 and BL3.

We categorize the applications into three classes, based on the impact of the increased capacity on their performance:

1. **Positive impact:** bzip2, gcc, hmmer, mcf, omnetpp, xalan, h264ref, astar and cactusADM. Most of them can be accelerated by 20%. The small MPKI reduction (2.7 to 0.8) in hmmer corresponds to three times fewer LLC misses, justifying its high speedup.
2. **Negative impact:** The performance of libquantum, lbm and milc deteriorates as the LLC size increases, likely due to the higher hit time as their MPKI is negligibly improved.

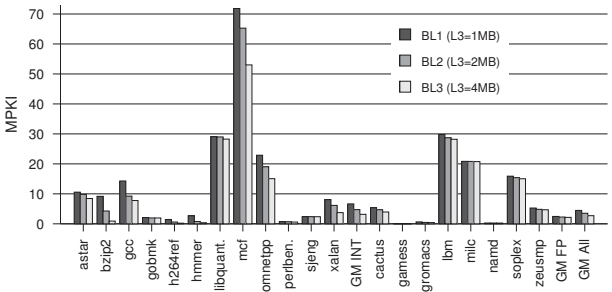

Figure 8: LLC size impact on MPKI for the baselines.

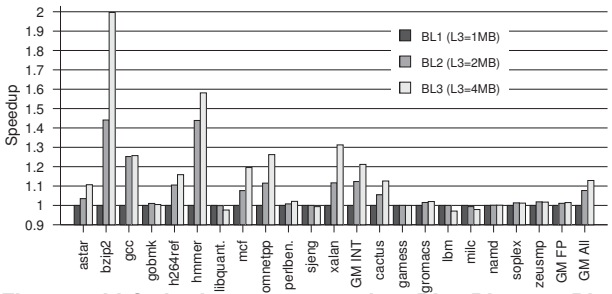

Figure 9: LLC size impact on speedup: BL2, BL3 over BL1.

3. **Slight positive or no impact:** The rest of the applications are insensitive to the increased cache size.

We conclude that only the first class of applications can gain from using compression. However, we use all the applications in our evaluation to verify that compression does not hurt the performance of insensitive applications.

Compression ratio (CR): Figure 10 depicts the compression ratio of our SC^2 scheme, FPC [2] and BDI [19]. Interestingly, most of the applications that have good CR belong to the class that would benefit from an increased cache size. Integer applications are compressed by 2.4X, on average (using geometric mean (GM)), while we conjecture that the mediocre compressibility of FP applications (less than 2X on average, except for soplex and zeusmp) is because FP data do not introduce redundancy to the same degree as integer data or because their types do not match the used 4-byte granularity in VFT.

While the CR of FPC and BDI is limited to less than 2X for most applications, SC^2 goes beyond 3X for several of them (mcf, omnetpp, xalan, gcc, gobmk). The higher CR of our scheme warrants more tags than prior works. When selecting 2x more tags instead of 4x, the average CR degrades by only 4% for FPC and BDI, while in SC^2 the CR degradation reaches 20%, on average. Exceptional cases in BDI are bzip2 and cactusADM with 15% and 11% CR reduction respectively,

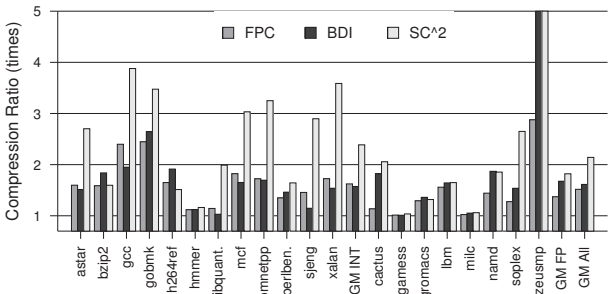

Figure 10: Compression ratio (LLC: 1MB) of SC^2, FPC and BDI.

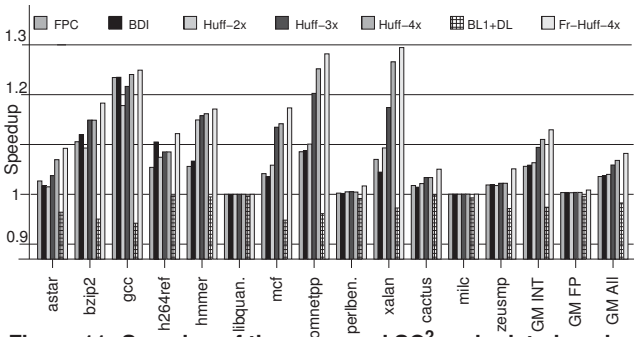

Figure 11: Speedup of the proposed SC^2 and related works.

and gcc (9% CR reduction) in FPC. Based on these findings, in the rest of evaluation we explore designs with up to 4x more tags in SC^2 and up to 2x for prior works. The latter design choice corroborates with the results presented in those studies.

Impact on MPKI: The high CR of SC^2 aids in reducing the L3 misses. MPKI is especially improved by 30% in gcc, mcf, omnetpp and hmmer, 48% in xalan, 15% in bzip2 and astar, and slightly or not improved in the rest of benchmarks.

Impact on speedup: Figure 11 shows the speedup achieved by SC^2 versus prior work, normalized to BL1. For clarity and due to space limitations, we only show perlbench, libquantum and milc from the applications that did not show any difference in speedup. Figure 11 has two additional bars that correspond, respectively, to an ideal SC^2 configuration where decompression is for free (Fr-Huff-4x), and a pessimistic baseline in which L3 cache hits pay an extra penalty, equivalent to the decompression latency (BL1+DL). These experiments establish upper and lower performance bounds, which will help us to evaluate the advantages of SC^2. With BL1+DL, we aim to show how much of the increase in L3 latency can be hidden by the out-of-order engine, while Fr-Huff-4x shows the potential benefits of the increased capacity achieved by SC^2 without paying any extra hit latency for decompression.

A significant speedup is noticed for the Positive-Impact applications such as omnetpp, xalan, gcc and mcf due to their reduced MPKI. Surprisingly, when comparing Figures 11 and 9, we notice that Huff-4x approximates the performance of BL3, as its improvement over BL1 is 27% in xalan, 25% in omnetpp and gcc, and 15% in mcf. The mean tag use of 85% (omnetpp), 90% (xalan), 68% (mcf) and 60% (gcc) indicates high utilization of the released cache space or, in other words, an increase in the effective cache capacity.

Interestingly, hmmer uses only 60% of the BL1 LLC space (region access). Despite its low CR, SC^2 can fit 5K more blocks in this region on average, reducing the LLC misses by 30% and improving the execution time by 15%. On the other hand, bzip2 is improved by 15% utilizing only 35% of the extra tag space. CactusADM and zeusmp are the only FP applications that enjoy some speedup.

Looking at bars BL1+DL and Fr-Huff-4x (Figure 11), we notice that most applications that benefit from statistical compression are sensitive to longer LLC hit latency, as shown by BL1+DL. However, Fr-Huff-4x shows how the performance

gains that accrue from fewer off-chip misses in SC^2 largely compensate for the longer hit latency. The rest of the applications neither benefit from nor are harmed by statistical compression, for the following reasons: a) compression is not sufficient meaning most blocks are saved uncompressed (gamess, gromacs, milc), and thus their hit latency or cache space is not affected, or b) their LLC miss rates are over 98% despite compression (lbm, libquantum, sjeng), or c) the LLC is accessed as rarely as once every 1000 instructions (gobmk, perlbench, namd); as a result, the longer hit latency is hidden. We also explore the scenario of always accounting 14 cycles for decompressing a cache block. We observed less than 1% degradation in speedup and only for the benefited applications.

Comparing SC^2 with FPC and BDI, we notice that it outperforms them for most memory-intensive workloads, while for the rest of the applications SC^2 performs similarly to them. The speedup of our SC^2 over FPC and BDI is 18% for xalan, 17% for omnetpp, 10% for mcf, 10% for hmmer and up to 6% for astar. On the other hand, BDI is marginally better (1%) in h264ref due to its higher CR, while in gcc SC^2 performs similarly to FPC and BDI, but it uses twice more tags. Even when 4x more tags are used in FPC and BDI, the gain in performance is marginal (1%) comparing to 2x, except for bzip2 (speedup of 22% against 12% – with BDI).

Bottomline: The proposed SC^2 manages to mimic the performance benefits of a 3X-4X larger cache for memory-intensive applications. In addition, the overhead of decompression has virtually no impact on performance for applications that are insensitive to larger cache capacity.

Impact on memory bandwidth utilization: Efficient compression in the LLC yields off-chip bandwidth utilization benefits due to the reduced number of transfers from main memory. Especially for the benefited applications, the bandwidth requirement is significantly reduced. In particular, the bandwidth demands for gcc are improved by 70% (800 fewer BPKI), in xalan by 45%, in bzip2 by 20%, in omnetpp by 20% (500 fewer BPKI) and in mcf by 18% (1000 fewer BPKI!). For the rest of applications, the reduction is 5-10% on average.

Impact of prefetching on compression: Up until now, prefetching was disabled since we aimed at exploring the pure impact of compression. Previous work [1, 3] has observed a positive interaction between compression and prefetching. When stride prefetching [13] is turned on (stride degree of 6), SC^2 consistently performs better than the baseline with prefetching enabled. For example, the improvement is 11.5% for xalan, 8% for mcf, 7.5% for gcc and 5.5% for bzip2. On the other hand, omnetpp, which does not benefit at all from stride prefetching, continues to have 15% to 20% improvement by the proposed Huffman-based compression. Finally, lbm enjoys 7.5% speedup with prefetching and compression, whilst libquantum and milc slightly degrade by less than 2%.

Code transition: Transition between encodings is supported by the mechanism described in Subsection 3.7. According to Figure 2, it is rarely needed within execution phases of

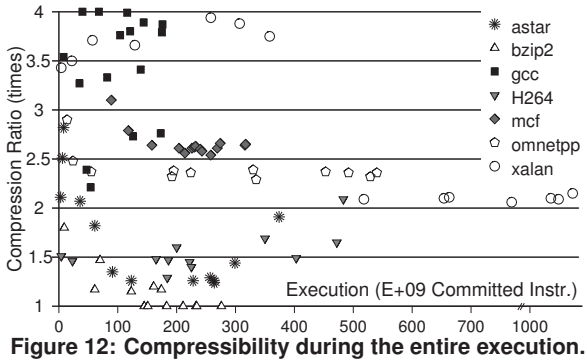

Figure 12: Compressibility during the entire execution.

10^9 instructions. We further verify this claim by measuring the CR for the entire execution of each application. We compute the encoding in the first simpoint and then use it in the remaining 10-15 simpoints (250M inst. each) that spread across the execution of each application. Figure 12 depicts the observed CR for some representative applications. Surprisingly, mcf, omnetpp, xalan and gcc remain highly compressible during their execution without changing the encoding. In xalan, the CR is close to 4X for 500 billion instructions, then drops and remains close to 2X until the end. In cactusADM, the CR is always 2X for 2.38×10^{12} instructions. On the other hand, in astar (perlbench and soplex as well) a new sampling may be required after executing 120 billion instructions. In bzip2 (gromacs and lbm), code transition occurs more frequently. Finally, the CR is consistently close to 1.5X in H264ref (libquantum and namd) and close to 1X in hmmer, gamess and milc. Overall, the same encoding can be used on the order of several seconds of execution with small degradation in CR.

5.2. Multi-core Evaluation

Our single-core evaluation highlights that SC^2 can speed up memory-intensive applications without hurting non-intensive ones. In a multi-core, the LLC is typically shared and thus applications running on different cores compete for it. Next, we evaluate SC^2 on the shared L3 cache of an 8-core processor, which runs multiprogrammed workloads (a different program per core) created by mixing 8 applications from SPEC2006. The application mix is created according to two criteria: 1) impact of compression on speedup, and 2) cache compressibility. A similar methodology is used by BDI [19].

The classification of the workloads based on the first criterion has already been discussed in the beginning of Section 5. As for compressibility, the applications are divided into three groups using Figure 10: Low (LC: milc, gamess, gromacs and hmmer), Medium (MC: bzip2, h264ref, perlbench, lbm and namd) and High (HC: the rest). We have created 15 multiprogrammed workloads (see Table 4) by combining applications using the two classifications. Based on the impact on speedup, we divide them in three groups: Negative or no impact (N), mixed impact (M), positive impact only (P). Then, in every group we use all possible combinations based on cache compression. We avoid including the same application many times in the same mix except for N-LCMC, N-HC and P-HC.

Table 4: Multiprogrammed workloads.

Compress.	LC: Low ($< 1.5X$); HC: High ($\geq 2X$); MC: Med; LMHC: All.
Name	Applications
Negative or No impact (N)	
N-LCMC	gamess-gromacs-milc-milc-lbm-perlbench-namd-perlbench
N-LCHC	gamess-gromacs-milc-soplex-libquantum-gobmk-sjeng-zeusmp
N-MCHC	lbm-perlbench-namd-libquantum-gobmk-sjeng-soplex-zeusmp
N-HC	libquantum-gobmk-sjeng-soplex-zeusmp-gobmk-libquantum-zeusmp
Mixed impact (M)	
M-LCMC	h264ref-hmmer-bzip2-milc-lbm-perlbench-gromacs-namd
M-LMHC1	bzip2-astar-omnetpp-gcc-gamess-gobmk-perlbench-zeusmp
M-LMHC2	cactusADM-bzip2-xalan-omnetpp-gamess-milc-libquantum-lbm
M-LMHC3	mcf-gcc-bzip2-milc-lbm-gromacs-gobmk-h264ref
M-LMHC4	xalan-astar-mcf-namd-gobmk-sjeng-gamess-lbm
M-HC1	omnetpp-xalan-mcf-libquantum-gobmk-soplex-sjeng-gcc
M-HC2	mcf-omnetpp-gcc-xalan-cactusADM-astar-libquantum-soplex
Positive impact (P)	
P-LMHC1	hmmer-bzip2-h264ref-cactusADM-astar-xalan-gcc-omnetpp
P-LMHC2	hmmer-bzip2-h264ref-mcf-xalan-gcc-omnetpp-astar
P-MCHC	bzip2-h264ref-cactusADM-astar-xalan-gcc-omnetpp-mcf
P-HC	xalan-gcc-omnetpp-mcf-cactusADM-astar-xalan-omnetpp

We run these workloads on the multi-core baseline configurations (MCBL1/2/3 – Table 2) and the Huffman-based SC^2 ones (MC-Huff-2x/3x/4x – Table 3) where the compressed cache is a 4-MB LLC (like MCBL1). We compare them against a similar system with a 4-MB LLC compressed with FPC and BDI. Figure 13 illustrates the weighted speedup for all the simulated systems. Results are normalized to MCBL1. Overall, SC^2 improves performance by 10% over MCBL1, surprisingly more than MCBL2 and MCBL3 despite their doubled and quadrupled physical LLC size, respectively. Moreover, it outperforms FPC and BDI thanks to the superior compression ratios and despite the longer decompression latency.

A closer look reveals the following. Starting from the workloads that do not gain from compression (N-), SC^2 shows both gains and losses. In N-LCMC, the improvement is because in lbm compression and prefetching have a beneficial interaction, while the rest remain unaffected. In N-MCHC, the losses in libquantum are compensated by the gains in lbm, while MCBL3 degrades due to the extra 17 cycles in hit latency. On the other hand, SC^2 reduces performance by 2% in N-HC because libquantum and probably zeusmp (each has two instances) are penalized by the extra decompression latency, as opposed to BDI that keeps performance unaffected.

In the Mixed impact workloads, we notice that as we go from LCMC to LMHC and eventually to HC, the performance gap between SC^2 and FPC, BDI or MCBL2/3 opens up: As the number of applications with high CR increases, more resources are freed up to the benefit of the cache-intensive workloads. This compensates for the extra decompression latency, although there are also applications that do not gain from compression. Moreover, the presence of at least two instances of gcc, bzip2 and h264ref where FPC and BDI perform close to SC^2, justifies the small performance gap in LCMC, LMHC1 and LMHC3. On the other hand, in M-LMHC2 where there are 5 workloads with positive impact (e.g., xalan and omnetpp) the speedup over FPC and BDI is 10%.

Finally, in memory-intensive workloads (P-), we notice that SC^2 enjoys a speedup of 20%, on average, and up to 35% over MCBL1. Moreover, it surpasses the performance of MCBL3 by 7%, on average. The latter is attributed to the fact that MC-Huff-4x provides similar cache capacity to MCBL3 but with smaller impact on the LLC hit latency. Especially, in the P-HC workload, the 15% performance improvement over FPC and BDI suggests that SC^2 is really competitive in memory-intensive workloads that are also highly compressible.

Table 5: LLC leakage power and dynamic energy per access.

Unit	Leakage power [mW]	Dynamic energy [nJ]
MCBL1	240	0.61
MCBL2	472	0.85
MCBL3	944	1.12
MC-Huff-2x	295	0.612
MC-Huff-3x	387	0.615
MC-Huff-4x	509	0.615
Compressor / VFT	8.4	0.144
Decompressor	0.1	0.148

SC^2 uses 30% more memory than a 4-MB baseline due to tag overheads (Table 1), extra (de)compressors and the VFT, but as it approaches the performance of a 4X larger cache, SC^2 is $\frac{4}{1.3} = 3.07X$ more efficient than such a cache. On the other hand, BDI has 20% overhead (4-b encoding, 10-b index and 16-b mask per tag) but may approach the performance of a 2X larger cache being only $\frac{2}{1.2} = 1.66X$ more efficient than that.

In this analysis of multi-core systems, we assume one running thread per core that corresponds to a different application. If multiple threads run on a single core, then in a context switch if the new thread belongs to one of these 8 applications, no code generation is required based on the fact that value locality varies little over time. Moreover, the coding was generated based on one VFT (of double size) across all applications of each multiprogrammed workload, showing that sampling is efficient both across time and applications.

In summary, we have shown that the proposed SC^2 scheme, thanks to its substantially higher compression ratio, can yield better performance than simpler schemes despite the higher decompression latency it introduces.

5.3. Power/Energy Analysis

Previous analysis shows that SC^2 can improve the performance of multi-core systems by increasing LLC utilization. However, performance is not the only design goal as the power budget is a constrain. This section presents a power/energy analysis of the proposed scheme using the setup of the multi-core system.

Using CACTI in 32-nm technology, the tag array is modeled using low power transistors (ITRS-LOP), while the data array that dominates the cache is modeled using low standby power transistors (ITRS-LSTP) to keep the leakage power at low levels. The compressor and the VFT are modeled using high performance transistors (ITRS-HP), while the decompressor is implemented in Synopsys using low power cells.

Table 5 summarizes the dynamic energy (nJ) per access and the leakage power (mW) of the baseline and compressed LLC (the leakage power refers to all the cache banks of the LLC). It also provides the leakage power of a compressor, a VFT and a

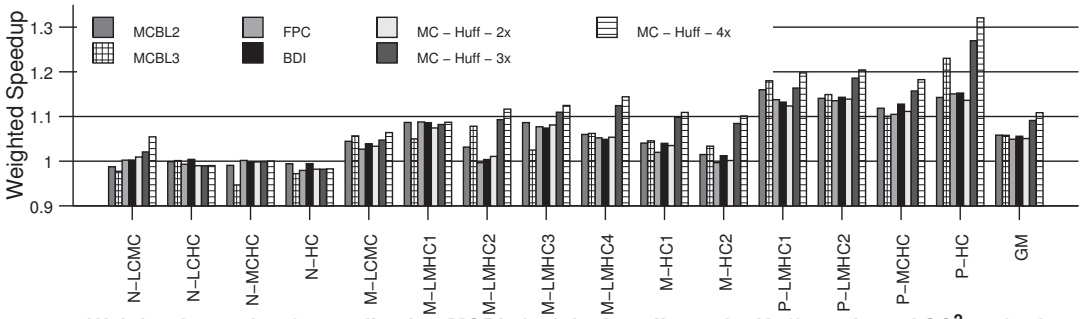

Figure 13: Weighted speedup (normalized to MCBL1) of the baselines, the Huffman-based SC^2 and prior arts.

decompressor, as well as the dynamic energy for compressing a cache block or updating the counters of the VFT, and the dynamic energy to fully decompress a compressed block. The MC-Huff-2x/3x/4x cache structures have different leakage power and slightly different dynamic energy because, despite having the same physical data store, their tag-store size varies.

Using the leakage power and dynamic energy of Table 5 and the collected statistics of the simulation (#misses, #hits, simulation time), we can calculate the total energy consumption of the LLC in the baseline and the corresponding configurations with compression enabled. Our energy analysis models the consumed energy of: (1) the accessed tags and data for the hits and for writing the missed blocks; (2) the accessed tags for the misses; (3) the dynamic energy due to the decompressions of read and evicted dirty blocks; (4) the compression attempts of the missed or modified blocks; (5) VFT updates during sampling; (6) the accessed tags and data of shifted blocks due to compaction (accessed twice per shifted block). The first two apply for both baseline and configurations with compression enabled, while (3)-(6) apply only to the ones with compression on. Finally, the total leakage power of the tag store, the data store, the eight compressors, the sixteen decompressors (twice more for code transition), and the VFT is multiplied with the total simulation time to derive the total static energy.

The total energy overheads of MC-Huff-2x and MC-Huff-4x is 53% and 55% lower than using a two times (MCBL2) or four times (MCBL3) physically larger LLC respectively. In particular, the energy overheads for the MC-Huff-2x and MC-Huff-4x are lower than the respective MCBL2 and MCBL3 by 65% and 61% for P-workloads, 56% for M-workloads, and 26% and 46% respectively for the N-workloads.

The total consumed energy is dominated, 75-80%, by static energy (due to leakage power) rather than dynamic energy. The 8 compressors, 16 decompressors, and one VFT contribute to the total static energy by only 22% (MC-Huff-2x) and 15% (MC-Huff-4x); most static energy is due to larger tag store. On the other hand, 70% of total dynamic energy is due to compaction to release contiguous space. Decompression consumes more energy than compression in the P-impact workloads, while the opposite is noticed in the N-impact ones.

As SC^2 reduces LLC misses, this increases performance and can potentially result in significantly improved energy efficiency, if the gain in energy is larger than the overheads due to compression, decompression and larger tag store. Using a DDR3 memory with 80pJ energy per bit [17] the energy to access a 64-byte block is approximated to 40nJ. We compute energy savings by multiplying this by the number of avoided misses thanks to compression. We compare the total energy benefit, which factors in both gains and overheads, of SC^2 and baseline configurations in a multi-core system. We find that a Huffman-based SC^2 is 17% (MC-Huff-2x) and 43% (MC-Huff-4x) more energy efficient than a two times (MCBL2) and four times (MCBL3) physically larger cache, respectively.

Bottomline: When the LLC is augmented with Huffman-based compression, the energy overheads introduced, due to compression, decompression, compaction and the extra tag stores, are less than 50% than using a 4X larger physical cache.

6. Related Work

Other work on cache compression includes the dictionary-based approach proposed by Hallnor and Reinhardt [10] where a cache block is compressed using the LZSS algorithm [25] on the LLC. The reported compression ratio is typically less than 2X. Other work has exploited specific aspects of value locality. Islam and Stenstrom [14] accelerate loads that return the value 0 using the Zero Value Caches (ZVC). Dusser et al. compress null blocks with the Zero-Content Augmented caches (ZCA) [8]. The Frequent Value Compression (FVC) [28] uses a small set (4 to 8) of statically determined frequent values to squeeze two blocks into the same blockframe. Residue cache [15] builds on FPC to reduce cache size and power. Baek et al. [5] study size-aware insertion and replacement techniques of compressed blocks in the LLC that are orthogonal to SC^2. By contrast, this paper explores the potential and the challenges of statistical compression caches which, as we have shown, yield a substantially higher compression ratio.

7. Conclusions

Unlike previous work on cache/memory compression, which opt for simple schemes to keep the decompression latency low, the goal of this study is to explore the consequences of using aggressive statistical compression to envision substantially higher cache utilization than what has been achieved before. Huffman encoding, as opposed to simple pattern-based compression schemes, is the key approach taken in this study.

A first observation is that SC^2 offers substantially better compressibility than simple compression approaches attempted in the past. We show that the proposed Huffman-based SC^2 yields a compression ratio of 2.2X, on average, as compared to 1.5X for simple pattern-based compression schemes proposed in the past. While this offers an opportunity to virtually enlarge the cache capacity at a similar rate, the key issue with statistical compression is the overhead it imposes on value statistics acquisition, compression and decompression.

This study shows that the overhead of statistics acquisition can be afforded because value locality is stable enough in both time as well as application dimensions. This paper shows that sampling of value frequency yields a stable compression factor for a single application running for at least a second suggesting that sampling is needed very rarely. In addition, we show that value frequency acquisition across several applications can be practically used to achieve high compressibility. Importantly, we find that while compression latency can be kept off of the critical memory-access path, decompression latency does not impact adversely on any of the applications in this study.

Finally, we observe that by employing statistical compression it is possible to enlarge the cache capacity by a factor between 3X-4X with modest hardware investments (about 30% area for additional tags and metadata). Specifically, this study shows how a 4-MB shared LLC can be extended with SC^2 to yield the same speed improvement, on average, as a 4X larger cache but with about 50% less power dissipation.

8. Acknowledgements

We acknowledge Rubén Titos Gil and Anurag Negi (now affiliated with NVIDIA) for their invaluable help and comments during the writing of the manuscript. We would also like to thank the anonymous reviewers for their insightful feedback on earlier versions of this paper. This work was funded by the Swedish Research Council under the CHAMPP project (621-2009-4566) and by the Swedish Foundation for Strategic Research (SSF) under the SCHEME project (RIT10-0033). The simulations ran on the resources provided by the Swedish National Infrastructure for Computing (SNIC) at C3SE.

References

[1] A.-R. Adl-Tabatabai, A. M. Ghuloum, and S. O. Kanaujia, "Compression in cache design," in *Proceedings of the 21st annual international conference on Supercomputing*, ser. ICS '07. ACM, 2007.

[2] A. R. Alameldeen and D. A. Wood, "Adaptive cache compression for high-performance processors," in *Proceedings of the 31st annual international symposium on Computer architecture*, ser. ISCA '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 212–.

[3] ——, "Interactions between compression and prefetching in chip multiprocessors," in *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 228–239.

[4] A. Arelakis and P. Stenstrom, "A case for a value-aware cache," *IEEE Computer Architecture Letters*, vol. 99, no. RapidPosts, p. 1, 2012.

[5] S. Baek, H. G. Lee, C. Nicopoulos, J. Lee, and J. Kim, "ECM: Effective capacity maximizer for high-performance compressed caching," in *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*, 2013, pp. 131–142.

[6] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, 2011.

[7] X. Chen, L. Yang, R. P. Dick, L. Shang, and H. Lekatsas, "C-pack: A high-performance microprocessor cache compression algorithm." *IEEE Trans. VLSI Syst.*, vol. 18, no. 8, pp. 1196–1208, 2010.

[8] J. Dusser, T. Piquet, and A. Seznec, "Zero-content augmented caches," in *Proceedings of the 23rd international conference on Supercomputing*, ser. ICS '09. ACM, 2009, pp. 46–55.

[9] M. Ekman and P. Stenstrom, "A robust main-memory compression scheme," in *Proceedings of the 32nd annual international symposium on Computer Architecture*, ser. ISCA '05. IEEE Computer Society, 2005, pp. 74–85.

[10] E. G. Hallnor and S. K. Reinhardt, "A unified compressed memory hierarchy." in *HPCA*. IEEE Computer Society, 2005, pp. 201–212.

[11] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun, "Transactional memory coherence and consistency," in *Proceedings of the 31st Annual International Symposium on Computer Architecture*, ser. ISCA '04. IEEE Computer Society, 2004, pp. 102–.

[12] D. A. Huffman, "A method for the construction of minimum-redundancy codes," *Proceedings of the Institute of Radio Engineers*, vol. 40, no. 9, pp. 1098–1101, September 1952.

[13] S. Iacobovici, L. Spracklen, S. Kadambi, Y. Chou, and S. G. Abraham, "Effective stream-based and execution-based data prefetching," in *Proceedings of the 18th annual international conference on Supercomputing*, ser. ICS '04. ACM, 2004, pp. 1–11.

[14] M. Islam and P. Stenstrom, "Zero-value caches: Cancelling loads that return zero," in *Parallel Architectures and Compilation Techniques, 2009. PACT '09. 18th International Conference on*, 2009.

[15] S. Kim, J. Lee, J. Kim, and S. Hong, "Residue cache: a low-energy low-area L2 cache architecture via compression and partial hits," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-44 '11. ACM, 2011, pp. 420–429.

[16] S. Y. Larin, "Exploiting program redundancy to improve performance, cost and power consumption in embedded systems," Ph.D. dissertation, 2000, AAI3033677.

[17] K. Malladi, F. Nothaft, K. Periyathambi, B. Lee, C. Kozyrakis, and M. Horowitz, "Towards energy-proportional datacenter memory with mobile DRAM," in *Computer Architecture (ISCA), 2012 39th Annual International Symposium on*, 2012, pp. 37–48.

[18] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "Cacti 6.0: A tool to model large caches," HP Laboratories, Technical Report HPL-2009-85, 2009.

[19] G. Pekhimenko, V. Seshadri, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Base-delta-immediate compression: practical data compression for on-chip caches," in *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, ser. PACT '12. ACM, 2012, pp. 377–388.

[20] D. Salomon, *Variable-length Codes for Data Compression*, 2007.

[21] D. Sanchez and C. Kozyrakis, "Vantage: scalable and efficient fine-grain cache partitioning," in *Proceedings of the 38th annual international symposium on Computer architecture*, ser. ISCA '11. ACM, 2011, pp. 57–68.

[22] E. S. Schwartz and B. Kallick, "Generating a canonical prefix encoding," *Commun. ACM*, vol. 7, pp. 166–169, March 1964.

[23] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, ser. ASPLOS-X. ACM, 2002.

[24] A. Snavely and D. M. Tullsen, "Symbiotic jobscheduling for a simultaneous multithreaded processor," in *Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*, ser. ASPLOS IX. ACM, 2000, pp. 234–244.

[25] J. A. Storer and T. G. Szymanski, "Data compression via textual substitution," *J. ACM*, vol. 29, no. 4, pp. 928–951, Oct. 1982.

[26] M. Thuresson, L. Spracklen, and P. Stenstrom, "Memory-link compression schemes: A value locality perspective," *Computers, IEEE Transactions on*, vol. 57, no. 7, pp. 916 –927, july 2008.

[27] Y. Xie and G. H. Loh, "PIPP: promotion/insertion pseudo-partitioning of multi-core shared caches," in *Proceedings of the 36th annual international symposium on Computer architecture*, ser. ISCA '09. ACM, 2009, pp. 174–183.

[28] J. Yang, Y. Zhang, and R. Gupta, "Frequent value compression in data caches," in *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, ser. MICRO 33. ACM, 2000.